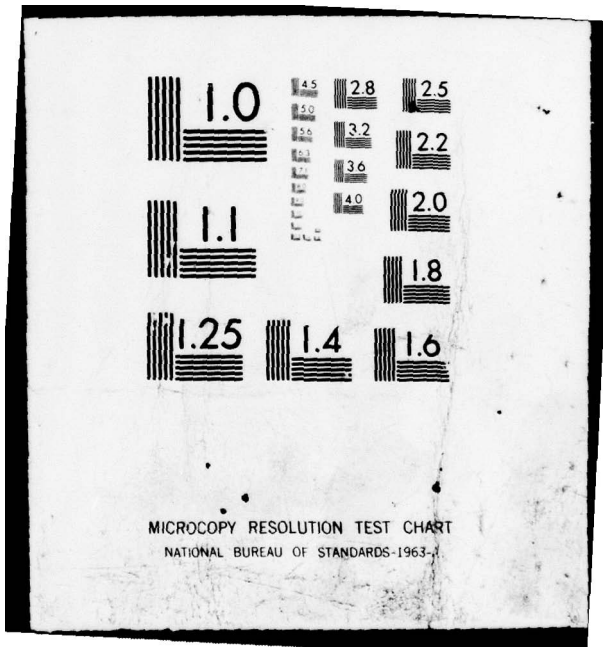
AD-A074 851 MARYLAND UNIV COLLEGE PARK COMPUTER SCIENCE CENTER F/G 20/6
ARCHITECTURE FOR HIGHER LEVEL DIGITAL IMAGE PROCESSING.(U)
OCT 78 T J WILLETT DAAG53-76-C-0138

UNCLASSIFIED NL

1 OF 1
AD A074 851

END
DATE FILMED
11-78
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

LEVEL II

AD A074851

ARCHITECTURE FOR HIGHER LEVEL DIGITAL IMAGE PROCESSING

October 30, 1978

This is the second quarterly status report on a program for Image Understanding Using Overlays, conducted by Westinghouse for Maryland under contract DAAG 53-76-C-0138 with the U.S. Army Mobility Equipment Research and Development Command, Ft. Belvoir, Va. 22060.

Prepared for

Computer Science Center
University of Maryland
College Park, Maryland 20742

By

Westinghouse Defense and Electronics Systems Center
Systems Development Division
Baltimore, Maryland 21203

(6) ARCHITECTURE FOR HIGHER LEVEL DIGITAL IMAGE PROCESSING.

(11) 30 Oct[redacted]978 (12) 35

This is the second quarterly status report on a program for Image Understanding Using Overlays, conducted by Westinghouse for Maryland (15) under contract DAAG 53-76-C-0138 with the U.S. Army Mobility Equipment Research and Development Command, Ft. Belvoir, Va. 22060.

(9) Quarterly status rept. no. 2,

Prepared for

Computer Science Center
University of Maryland
College Park, Maryland 20742

(10) Thomas J. Willett

By

Westinghouse Defense and Electronics Systems Center
Systems Development Division
Baltimore, Maryland 21203

403 018

TABLE OF CONTENTS

INTRODUCTION

This is the second quarterly status report on a program to investigate various approaches to the design of architecture for higher level digital image processing algorithms being conducted by the Westinghouse Systems Development Division for the Computer Science Center, University of Maryland. This two-year program is a continuation of a program entitled "Algorithms and Hardware Technology for Image Recognition", which was initiated in 1976. The report was prepared by Mr. Thomas J. Willett in consultation with Messrs. George McAfee, John Murtha, Oscar Cromer, and David Joy. The Westinghouse program manager is Dr. Glenn Tisdale.

During the quarter, monthly technical meetings were held at Maryland which included representatives from the Army Night Vision Laboratory, the University of Maryland, and Westinghouse. Team members from NVL were Dr. George Jones, Mr. John Dehne, and Mr. Peter Raimondi, and from the University of Maryland, Profs. David Milgram and Azriel Rosenfeld.

This report begins with a continuation of a description of bit slice microprocessors with the emphasis on control units this time, and an examination of commercially available units. Several more Maryland algorithms, namely non-linear probabilistic relaxation, connected components, and Superlink are described. Hardware implementations for non-linear probabilistic relaxation and connected components are described in the next section. The final section shows some tentative conclusions, in light of the above continuing analysis, regarding an appropriate architecture for image processing both for the image processing module and the array of modules.

1.0 CONTROL UNIT

In the first quarterly report, we described the general architecture of bit slice machines and gave an example of a macroinstruction (COMPARE AB) converted into a series of microinstructions. It may be worthwhile for the reader to review those notions before starting this section. We first describe a control unit in general and then move into a discussion of several commercially available fit slice control units.

1.1 General Description

In Figure 1-1, we show a functional block diagram of a control unit. The instruction register receives a macroinstruction from the memory via the data bus; to receive an instruction in just one machine cycle, the instruction width is only as wide as the data bus and the instruction register. The instruction register contains operation code (instruction or command to be performed) and the operands (data to be manipulated) such as the number of a selected register, a variable to be compared with a constant, or the address of an input/output port. The Mapping ROM begins the process of decoding the macroinstruction in the instruction register into microinstructions which directly control the computer's resources. The output of the Mapping ROM will be wider than the Operation Code in the instruction register and allows for a significant number of microinstructions per macroinstruction. That is, it allows for a greater range of starting address for the Microprogram ROM; typically the Mapping ROM width is 12 bits for an Operation Code width of 8 bits. The Microprogram Counter accepts the starting address from the Mapping ROM and points to the first microinstruction in the Microprogram ROM. This allows the address of the Microprogram ROM to be changed and its output to settle while the current microinstruction is being presented to the computer hardware from the Microprogram Register. The Microprogram Sequence Controller has two functions; it uses the output of the Test Condition

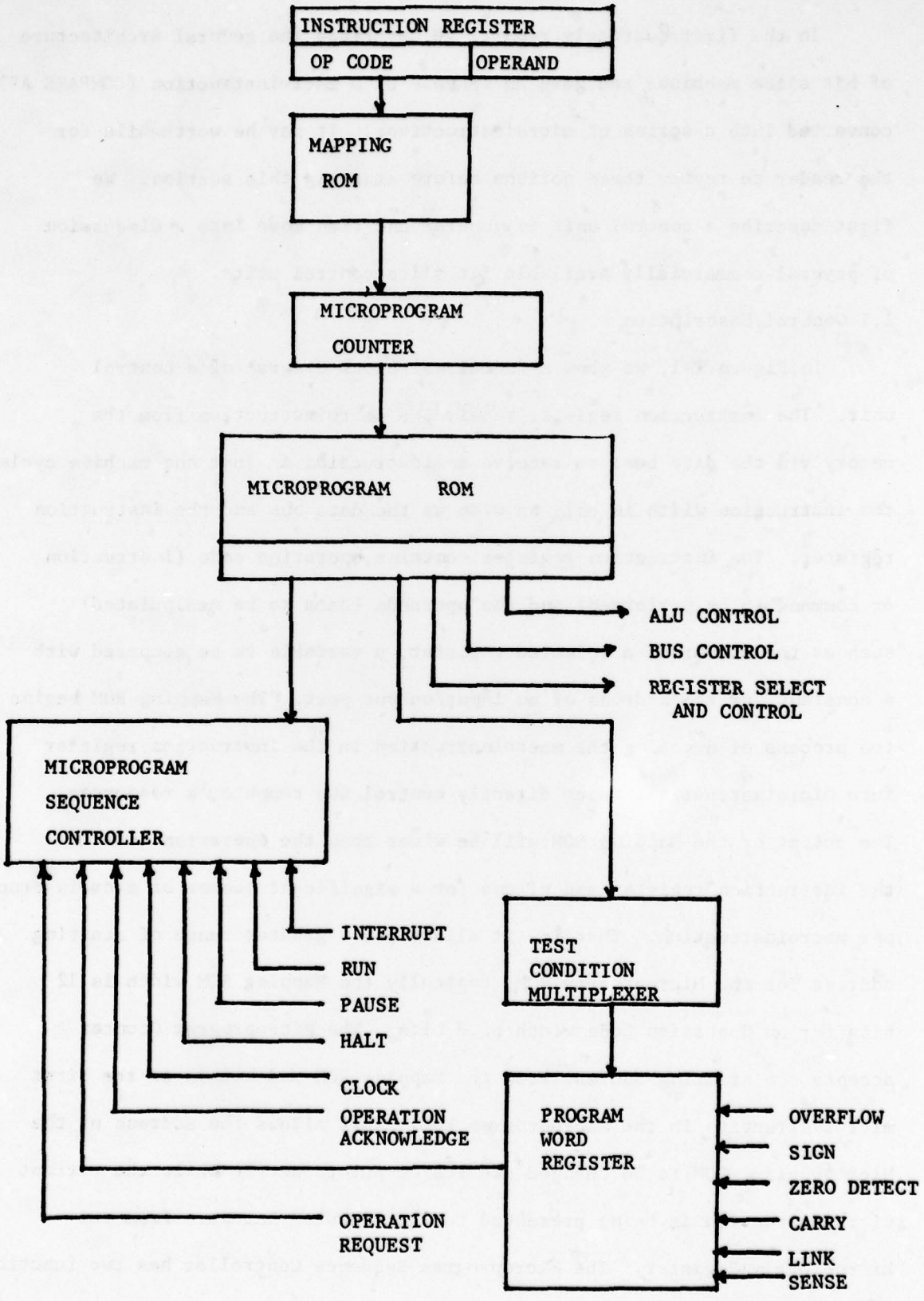


Figure 1-1. CONTROL UNIT

Multiplexer to determine if a branch, subroutine or jump condition should be done and it synchronizes events external to the control unit with the Control Unit such as RUN, HALT, and PAUSE. After every ALU (Arithmetic Logic Unit) operation or interrupt, the condition codes are loaded into the Program Status Word Register and presented to the Test Condition Multiplexer. If true, the output of the Test Condition Multiplexer will enable a branch instruction in the Microprogram Register.

1.2 Branches, Conditions, Subroutines, and Instruction Fetch

To execute a fixed sequence of microinstructions, the Control Unit need only look like Figure 1-2. The output of the Microprogram

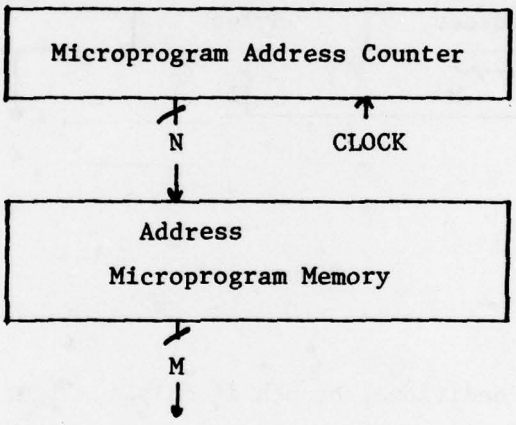


Figure 1-2. Control Unit With Counter

Address Counter is the next address of a microinstruction found in the Microprogram Memory. To update the address from 011 to 012, the clock pulse is used to update the counter. But this approach to the Control Unit does not allow a branch or jump to another microinstruction based on results obtained from the execution of the present microinstruction.

To execute a jump we must be able to load that address into the Microprogram Address Counter. This is accomplished via a control bit in the microprogram word format as shown in Figure 1-3, and allows for an N-way branch.

For example, if N = 8, then the branch or jump may go to any of 256 addresses before or after the current instruction.

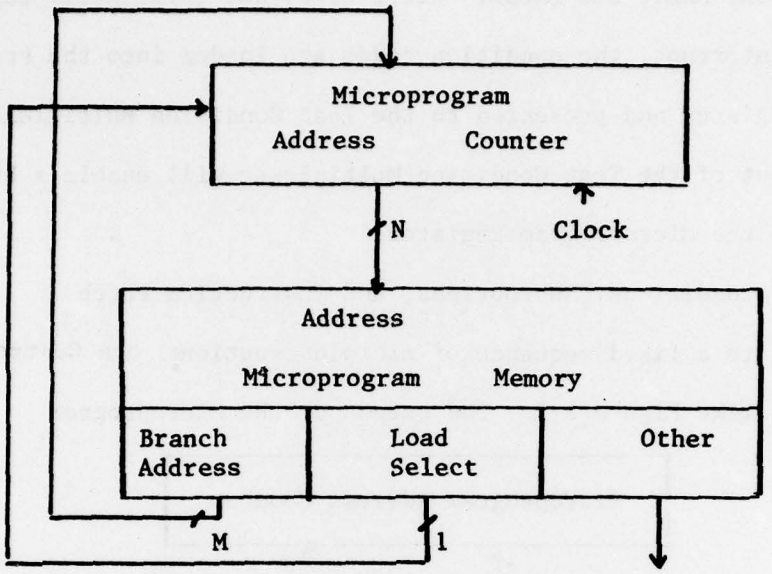


Figure 1-3

Setting up a conditional branch is only a slight variation on Figure 1-3, i.e. some sort of enable signal must be input to the Microprogram Address Counter to allow the Branch Address to be loaded. This is shown in Figure 1-4. Bits S_1 and S_0 are a two bit field which controls whether the instruction set is continued, jump is unconditional, jump on condition 1 true, or jump on condition 2 true. The bits D_1 and D_2 represent the conditions 1 and 2, e.g. if S_0 and S_1 are such that the jump on condition 1 is in effect and D_1 is binary 1, then the jump is made, and if D_1 is binary 0 then the jump is not made.

To overlap the microinstruction fetch procedure, a pipeline register contains the microinstruction currently being executed, while the address of the next microinstruction is applied to the Microprogram Memory and the contents of that memory word are being fetched and set up at the input to

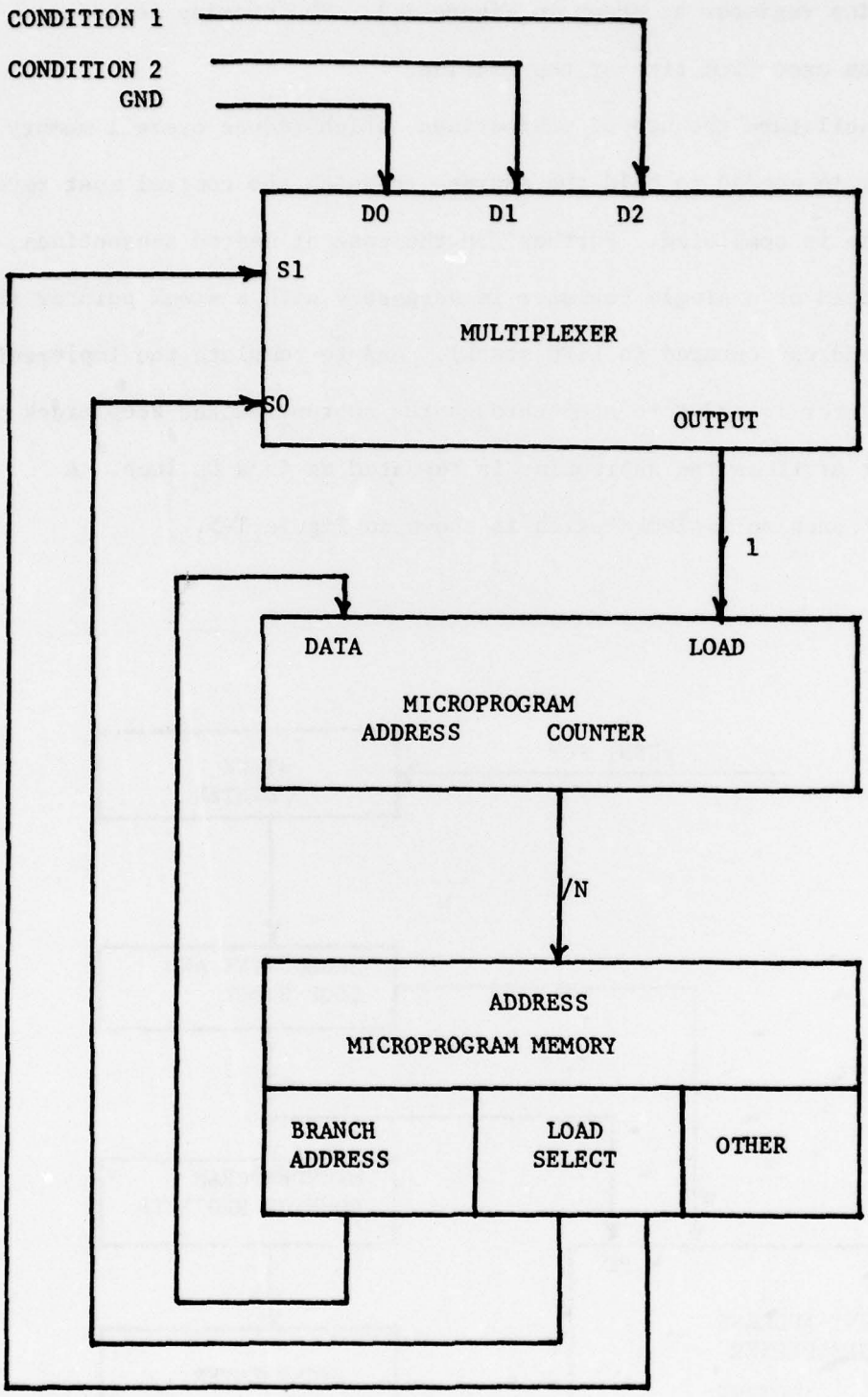


Figure 1-4. BRANCH ARCHITECTURE

the pipeline register as shown in Figure 1-1. The overlap technique reduces the execution time of the machine.

To facilitate the use of subroutines, which reduce overall memory size, a register is needed to hold the address to which the control must return once the routine is completed. Further, in the case of nested subroutines, a stack instead of a single register is necessary with a stack pointer showing the last address entered (a LIFO stack). And to complete the implementation, an incrementer is added to step through the subroutine and keep track of the number of times the subroutine is repeated as in a D_0 loop. A portion of such an implementation is shown in Figure 1-5.

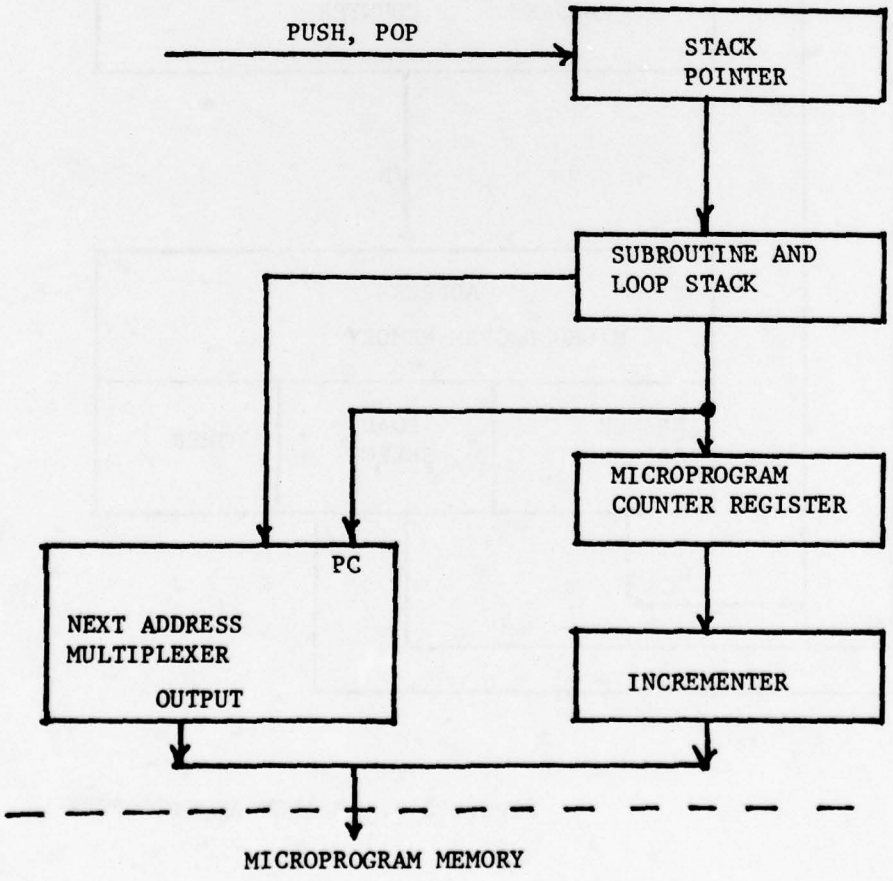


Figure 1-5. Subroutine Implementation

Thus far, we have discussed control units in general plus described implementations of the more important microinstructions in the sense that the others are variations of these. We now consider the several commercially available units and their comparative performance.

1.3 Commercially Available Units

The Control Units commercially available do not all exhibit the traditional cascading property of bit slice ALU's. Some really come in a single package which houses all the circuitry required to address anywhere from 512 to 4096 control memory words. The address space is increased by paging, i.e. where the memory is divided into blocks, each of which is equal to the length of the microcontroller's address space.

The Texas Instruments SN745482 has an internal full adder which allows relative addressing not found in any other microcontroller. It has subroutine capability and control lines to all the major components so that a field in the microinstruction may control each component directly. This allows more flexibility in the microinstruction set. However, the memory address port outputs are not three state which would be generally useful in automatic checkout of military designs with ground support equipment.

The Advanced Micro Devices AM2909 and AM 2911 are identical except that the 2909 offers four OR inputs to the subroutine incrementer and one more direct data input to the subroutine enable line. Both the 2909 and 2911 have no inputs for instructions but have a series of control lines which may be encoded by a microinstruction field.

The Motorola MC 10801 microcontroller is the fastest device in terms of clock frequency since it is built from ECL technology. Further, there are several architectural features which also contribute to increasing its speed. A repeat register is present allowing single microinstructions or subroutines to be repeated without the usual bookkeeping microinstructions,

thus increasing throughput. The Control Unit has microinstruction fetch also and the status bits information is stored internally for control as in the AMD 2911.

Monolithics Memories 67110 is a microprogram unit with an addressing capability of 512 words. There is a nice internal looping capability which increases throughput but the single level subroutine capability is small. The addressing scheme is based on the program counter as previously described, but a bit is reserved for branching which takes the control to the next even-odd pair; this simply means that the jump to instruction must be placed in the appropriate address with regard to the even-odd pair. There is internal storage for various flags and next instruction fetch.

The Intel 3001 is another microprogram control unit with a 512 word capability. There is no internal subroutine capability and internal storage is available for flags for conditional branching. The 512 word addressing scheme is unique in that a branch from any address is controlled by that address to a fixed subset of the address space.

The Fairchild 9408 is a 10 bit wide microprogram control unit which is slower than the TTL microcontrollers. There is a facility for storing flags to control conditional branching and the subroutine stack is 4 deep. There are 10 bits of branch address, the least significant 3 of which may be supplied from another source. This could help decode macroinstructions.

The AMD 2910 consists of three cascaded AMD 2911's for a bit width of 12 and has a programmed logic array (PLA) for instruction decoding on one chip. The subroutine stack is five words deep and basically there is more hardware to handle subroutines and DO loops both on entrance and exit; this results in an increase in throughput. Also, there are some instructions aimed specifically at ending loops in an efficient manner.

Tables 1-1 is a summary of the physical characteristics of the processors just described and Table 1-2 summarizes the computing characteristics of the processors.

Table 1-1. Physical Characteristics of the Control Units.

BIT-SLICE DEVICE	TECHNOLOGY	UNIT/ELEMENT	UNIT/ELEMENT WIDTH BITS	PINS AND PACKAGE	POWER DISSIPATION (MW)
TI SN745482	T^2L	E	4	20 DIP	473
AMD 2909	T^2L	E	4	28 DIP	420
AMD 2911	T^2L	E	4	20 DIP	420
MC 10801	ECL	E	4	48 QUIL	1738
MMI 67110	T^2L	U	9	40 DIP	835
INTEL 3001	T^2L	U	9	40 DIP	850
FAIRCHILD 9468	I^2L	U	10	40 DIP	650
AM 2901	T^2L	U	12	40 DIP	1024

Table 1-2. Computing Characteristics of Control Units

Bit-Slice Device	Fundamental Next-Address Mode	Subroutine Level	Internal Looping Capabilities	Internal Storage for Future Branch	Ports I	Ports O	Ports I/O	Control Lines Required
TI SN745482	Increment Program Counter	4	none	none	1	1	0	6
AMD 2909	Inc PC	4	none	next-address field	3	1	0	5
AMD 2911	Inc PC	4	none	next-address field	1	1	0	5
MC 10801	Inc PC	4	instruction, subroutine	status flags or next-address field	1	2	2	5+4+3
MMI 67110	Inc PC pairwise branch	1	program	status flags	1	1	0	8+2
INTEL 3001	Branch	4	none	status flags	1	1	0	7+4
Fairchild 9408	Inc PC	4	none	status flags	1	1	0	4-5
AMD 2910	Inc PC	5	program, instruction	next address	1	1	0	6

2.0 ALGORITHMS

This section provides a brief description of each algorithm; for a complete development and description, the reader is referred to the University of Maryland reports.

2.1 Relaxation

The discrete relaxation algorithm was described in the first quarterly report; we continue the relaxation development with the non-linear probabilistic case. Here probabilities are assigned to denote the possibility of an object being in a particular class. The strategy is to reinforce the probability $p_i(\lambda)$ of a given class for a given object a_i if other objects' labels, having high probabilities, are highly compatible with λ at a_i. On the other hand, $p_i(\lambda)$ should be decreased if other high probability labels are incompatible with λ at a_i. Further, low probability labels should have little effect on $p_i(\lambda)$ regardless of whether they are compatible with it. We may then set up a matrix of $p_i(\lambda_\ell)$'s and iterate the matrix by some function G, where $p_i^{k+1}(\lambda) = G = G[p_i^k(\lambda), r_{ij}(\lambda\lambda'), d_{ij}]$, according to the above strategy. The quantities $r_{ij}(\lambda,\lambda')$ and d_{ij} are the compatibility coefficients and weights, respectively. For our implementation, we shall assume that the $r_{ij}(\lambda,\lambda')$'s are the statistical correlation coefficients between objects a_i, class λ and objects a_j, class λ'.

The function $q_i^k(\lambda) = \sum_j d_{ij}[\sum_{\lambda'} r_{ij}(\lambda\lambda')\, q_j^k(\lambda')]$ has properties which follow the above strategy, i.e., if $p_j^k(\lambda')$ is high, and $r_{ij}(\lambda\lambda')$ is highly positive or negative, then $q_j^k(\lambda)$ reflects this. However, a small $p_j^k(\lambda')$ makes a relatively small contribution regardless of $r_{ij}(\lambda\lambda')$. In order to ensure that $p_i^{k+1}(\lambda)$ is non-negative, and that $\sum_\lambda p_i^{k+1}(\lambda) = 1$, we define

$$p_i^{k+1}(\lambda) = p_i^k(\lambda)[1 + q_i^k(\lambda)]/\sum_{\lambda} p_i^k(\lambda)[1 + q_i^k(\lambda)]$$

and again the strategy is obeyed.

2.2 Connected Components

The purpose of the algorithm is to segment the image data stream into smaller domains. Each small domain includes a single object in the image plane. This algorithm distinguishes between objects and isolates regions so that statistics for Classification Logic can be obtained.

Assume that the original image has been thresholded and the result is in binary form with gray levels exceeding g_1 shown as 1's in Figure 2-1a.

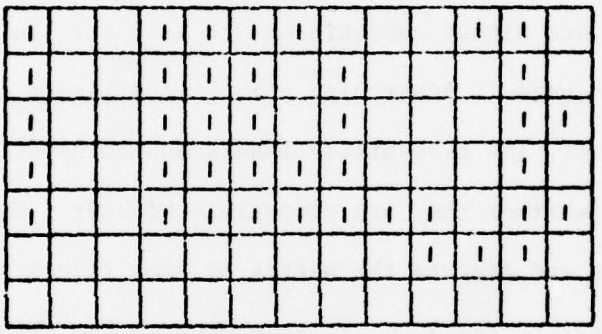


Figure 2-1a. Binary Image

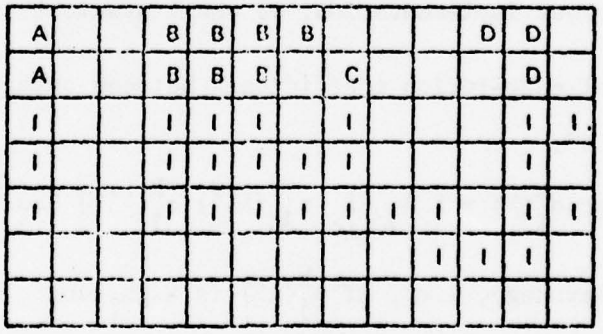


Figure 2-1b. Computations for Second Row

Two image lines are retained in memory so that each pixel can examine its neighbors to the left and above. No diagonal connections are permitted under this convention, and an adjacent (horizontal or vertical) pixel must be occupied in order to make a connection. No skips or gaps are

allowed, and the computations start one pixel in from the edge. In Figure 2-1b, there are four distinct regions, A, B, C, and D. The only possible connection between regions B and C is through a diagonal, which is not allowed. Computations for the fourth row are seen in Figure 2-1c.

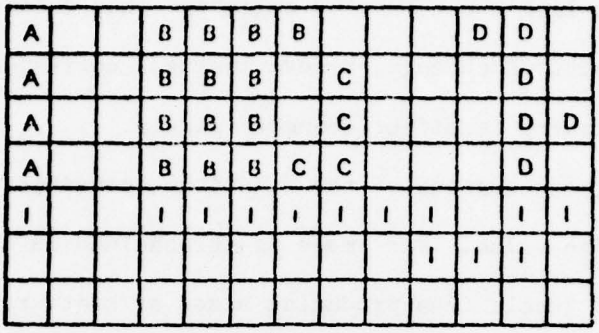


Figure 2-1c. Computations for Fourth Row

Here, there is a connection between regions B and C and an equivalence statement, B = C, is carried along. At the end of the sixth row, there is another connection between C and D (C = D) and all the regions are completed as seen in Figure 2-1d.

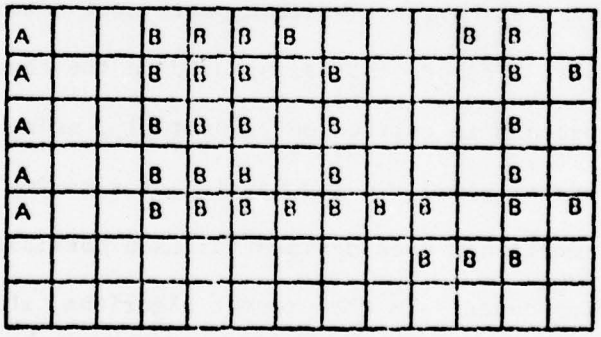


Figure 2-1d. Completed Image

The areas of A, B, C, and D are computed by cumulating the number of pixels assigned to each. The perimeter is calculated by cumulating the number of pixels assigned to each region which are neighbors of zeros, i.e., the neighbors did not exceed the gray level threshold, g_1.

2.3 Superlink

In the Gradient Operator algorithms gradients or edges are produced; in the Non-Maximum Suppression algorithm, these edges are thinned. Superlink is an attempt to link these thinned edges and thus delineate the perimeter of the object. Each edge, before linkage, corresponds to a single pixel, so that the linking in effect connects pixels.

To understand the mechanics of the algorithm, consider a pixel with an associated edge value. The image is thresholded in amplitude at a number of different levels thus producing a set of contours or perimeters of the object of interest within the image. We now consider a particular edge point and see if a perimeter passes through that pixel. If not, the edge point is discarded; if so, then that edge point is eligible for edge point linking. For each threshold, that edge point nearest the given edge point along the contour, i.e. an edge/perimeter point match exists there also, in the clockwise direction is recorded. Now this process is repeated for each of the thresholds, and that neighbor which appears most often is selected as the appropriate neighbor. The process is repeated in the counterclockwise direction. This process is carried out repeatedly, using each successive pixel as the new center pixel, in the fashion of a moving window.

A figure of merit has been devised for each possible linkage and is used in tie breaking cases. In the present algorithm, short straight paths are preferred as are those whose contrast does not vary much from one end to the other. The figure of merit turns out to be a weighted linear combination of straightness and contrast for the present analysis, although more work will be done by the University of Maryland under this contract.

3.0 HARDWARE IMPLEMENTATION OF ALGORITHMS

The purpose of this section is to provide hardware implementation designs for the algorithms described in Section 2.0 with an eye toward real time speeds.

The first case examined under non-linear probabilistic relaxation is (2x3) two classes and three objects. We then expand the problem. Yet to be considered is the interconnect problem and dynamic reconfiguration for a variable number of classes and objects, and reliability. It is important to consider special implementation for relaxation operations in order to provide for real or non-real time operations. The University of Maryland has estimated that it will require many hours to perform the relaxation computations for an image frame on a general purpose machine.

3.1 Non-Linear Probabilistic Relaxation

3.1.1 2x3 Case

For hardware implementation we first concentrate on q_i^k and expand it for the simple case of two classes $\lambda = \lambda_1, \lambda_2$ and three objects $i = 1, 2, 3$ as shown in Figure 3-1. We note several possible simplifications in the expansion, namely $r_{ii}(\lambda\lambda) = 1$ and $r_{ij}(\lambda\lambda') = r_{ji}(\lambda'\lambda)$. Replacing each of the correlation coefficients by capital letters A, B, ...0, i.e., $A = r_{11}(\lambda_1,\lambda_2) = r_{11}(\lambda_2,\lambda_1)$, $B = r_{21}(\lambda_1\lambda_1)$, ..., we can write $q_i^k(\lambda)$ as shown in Figure 3-2. Row 1 of each expression is composed of d_{11}, A, $p_1^0(\lambda_1)$ and $p_1^0(\lambda_2)$; the only difference is the relative position of the A coefficient. Similarly, rows 5 and 9 have the same structure. The same kind of remarks can be made about other row pairs, e.g., 2 and 4. In fact, Figure 3-3 shows the similarities. Because of the similarities in structure between rows, the same set of microinstructions, including rotation and register index, can form each side of each row. For example,

$$q_1^0(\lambda_1) = d_{11}[r_{11}(\lambda_1\lambda_1)\ p_1^0(\lambda_1) + r_{11}(\lambda_1\lambda_2)\ p_1^0(\lambda_2)] \quad (1)$$
$$+ d_{12}[r_{12}(\lambda_1\lambda_1)\ p_2^0(\lambda_1) + r_{12}(\lambda_1\lambda_2)\ p_2^0(\lambda_2)] \quad (2)$$
$$+ d_{13}]r_{13}(\lambda_1\lambda_1)\ p_3^0(\lambda_1) + r_{13}(\lambda_1\lambda_2)_3p^0(\lambda_2)] \quad (3)$$

$$q_1^0(\lambda_2) = d_{11}[r_{11}(\lambda_2\lambda_1)\ p_1^0(\lambda_1) + r_{11}(\lambda_2\lambda_2)\ p_1^0(\lambda_2)]$$
$$+ d_{12}[r_{12}(\lambda_2\lambda_1)\ p_2^0(\lambda_1) + r_{12}(\lambda_2\lambda_2)\ p_2^0\ (\lambda_2)]$$
$$+ d_{13}[r_{13}(\lambda_2\lambda_1)\ p_3^0(\lambda_1) + r_{13}(\lambda_2\lambda_2)\ p_3(\lambda_2)]$$

$$q_2^0(\lambda_1) = d_{21}[r_{21}(\lambda_1\lambda_1)\ p_1^0(\lambda_1) + r_2(\lambda_1\lambda_2)\ p_1^0(\lambda_2)] \quad (4)$$
$$+ d_{22}[r_{22}(\lambda_1\lambda_1)\ p_2^0(\lambda_1) + r_{22}(\lambda_1\lambda_2)\ p_2^0(\lambda_2)] \quad (5)$$
$$+ d_{23}[r_{23}(\lambda_1\lambda_1)\ p_2^0(\lambda_1) + r_{23}(\lambda_1\lambda\ )_2p_2^0(\lambda_2)] \quad (6)$$

$$q_2^0(\lambda_2) = d_{21}[r_{21}(\lambda_2\lambda_1)\ p_1^0(\lambda_1) + r_{21}(\lambda_2\lambda_2)\ p_1^0(\lambda_2)]$$
$$+ d_{22}[r_{22}(\lambda_2\lambda_1)\ p_2^0(\lambda_1) + r_{22}(\lambda_2\lambda_2)\ p_2^0(\lambda_2)]$$
$$+ d_{23}[r_{23}(\lambda_2\lambda_1)\ p_3^0(\lambda_1) + r_{23}(\lambda_2\lambda_2)\ p_3^0(\lambda_2)]$$

$$q_3^0(\lambda_1) = d_{31}[r_{31}(\lambda_1\lambda_1)\ p_1^0(\lambda_1) + r_{31}(\lambda_1\lambda_2)\ p_1^0(\lambda_2)] \quad (7)$$
$$d_{32}(r_{32}(\lambda_1\lambda_1)\ p_2^0(\lambda_1) + r_{32}(\lambda_1\lambda_2)\ p_2^0(\lambda_2)] \quad (8)$$
$$d_{33}[r_{33}(\lambda_1\lambda_1)\ p_3^0(\lambda_1) + r_{33}(\lambda_1\lambda_2)\ p_3^0(\lambda_2)] \quad (9)$$

$$q_3^0(\lambda_2) = d_{31}[r_{31}(\lambda_2\lambda_1)\ p_1^0(\lambda_1) + r_{31}(\lambda_2\lambda_2)\ p_2^0(\lambda_2)]$$
$$+ d_{32}[r_{32}(\lambda_2\lambda_1)\ p_2^0(\lambda_1) + r_{32}(\lambda_2\lambda_2)\ p_2^0(\lambda_2)]$$
$$+ d_{33}[r_{33}(\lambda_2\lambda_1)\ p_3^0(\lambda_1) + r_{33}(\lambda_2\lambda_2)\ p_3^0(\lambda_2)]$$

Figure 3-1. $q_i^k(\lambda)$ Expansion

$$q_1^0(\lambda_1) = d_{11}[p_1^0(\lambda_1) + A\ p_1^0(\lambda_2)] \quad (1)$$
$$+ d_{12}[B\ p_2^0(\lambda_1) + C\ p_2^0(\lambda_2)] \quad (2)$$
$$+ d_{13}[G\ p_3^0(\lambda_1) + I\ p_3^0(\lambda_2)] \quad (3)$$

$$q_1^0(\lambda_2) = d_{11}[A\ p_1^0(\lambda_1) + p_1^0(\lambda_2)]$$
$$+ d_{12}[F\ p_2^0(\lambda_1) + D\ p_2^0(\lambda_2)]$$
$$+ d_{13}[H\ p_3^0(\lambda_1) + O\ p_3^0(\lambda_2)]$$

$$q_2^0(\lambda_1) = d_{21}[B\ p_1^0(\lambda_1) + F\ p_1^0(\lambda_2)] \quad (4)$$
$$+ d_{22}[p_2^0(\lambda_1) + E\ p_2^0(\lambda_2)] \quad (5)$$
$$+ d_{23}[K\ p_3^0(\lambda_1) + N\ p_3^0(\lambda_2)] \quad (6)$$

$$q_2^0(\lambda_2) = d_{21}[C\ p_1^0(\lambda_1) + D\ p_1^0(\lambda_2)]$$
$$+ d_{22}[E\ p_2^0(\lambda_1) + p_2^0(\lambda_2)]$$
$$+ d_{23}[M\ p_3^0(\lambda_1) + J\ p_3^0(\lambda_2)]$$

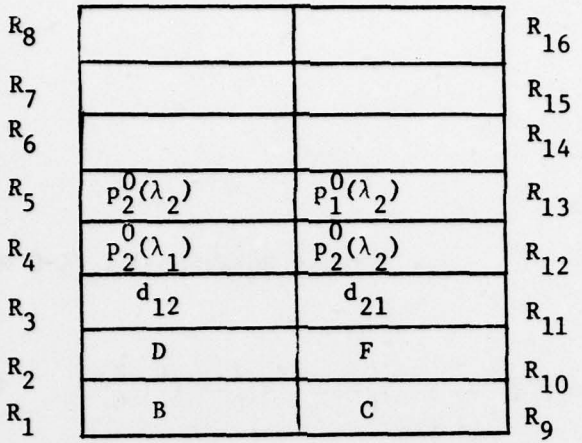
$$q_3^0(\lambda_1) = d_{31}[G\ p_1^0(\lambda_1) + H\ p_1^0(\lambda_2)] \quad (7)$$
$$+ d_{32}[K\ p_2^0(\lambda_1) + M\ p_2^0(\lambda_2)] \quad (8)$$
$$+ d_{33}\ [p_3^0(\lambda_1) + L\ p_3^0(\lambda_2)] \quad (9)$$

$$q_3^0(\lambda_2) = d_{31}[I\ p_1^0(\lambda_1) + O\ p_2^0(\lambda_2)]$$
$$+ d_{32}[N\ p_2^0(\lambda_1) + J\ p_2^0(\lambda_2)]$$
$$+ d_{33}[L\ p_3^0(\lambda_1) + p_3^0(\lambda_2)]$$

Figure 3-2. Collecting Terms

ROWS 1, 5, 9	A, E, L, $p_1^0(\lambda_1)$, $p_1^0(\lambda_2)$, $p_2^0(\lambda_1)$, $p_2^0(\lambda_2)$, $p_3^0(\lambda_1)$, $p_3^0(\lambda_2)$, d_{11}, d_{22}, d_{33}
ROWS 2, 4	D, F, C, B $p_2^0(\lambda_1)$, $p_2^0(\lambda_2)$, $p_1^0(\lambda_1)$, $p_1^0(\lambda_2)$, d_{12}, d_{21}
ROWS 3, 7	G, I, H, O $p_3^0(\lambda_1)$, $p_3^0(\lambda_2)$, $p_1^0(\lambda_1)$, $p_1^0(\lambda_2)$, d_{13}, d_{31}
ROWS 6, 8	K, M, N, J $p_3^0(\lambda_1)$, $p_3^0(\lambda_2)$, $p_2^0(\lambda_1)$, $p_2^0(\lambda_2)$, d_{23}, d_{32}

Figure 3-3. Row Similarities



move $R_9R_{12} \rightarrow R_6R_{14}$	$p_2^0(\lambda_1)\ p_2^0(\lambda_2)$
move $R_9R_{12} \rightarrow R_7R_{15}$	$p_2^0(\lambda_1)\ p_2^0(\lambda_2)$
mult $R_3R_6 \rightarrow R_6$	$d_{12}^2\ p_2^0(\lambda_1)^2$
mult $R_3R_{14} \rightarrow R_7$	$d_{12}\ p_2^0(\lambda_2)$
mult $R_3R_7 \rightarrow R_7$	$d_{12}\ p_2^0(\lambda_2)$
mult $R_3R_{15} \rightarrow R_{15}$	$d_{12}p_2^0(\lambda_1)$
mult $R_1R_6 \rightarrow R_6$	$B\ d_{12}p_2^0(\lambda_1)$
mult $R_9R_{14} \rightarrow R_{14}$	$C\ d_{12}p_2^0(\lambda_2)$
add $R_6R_{14} \rightarrow R_{14}$	$B\ d_{12}p_2^0(\lambda_1) + C\ d_{12}p_2^0(\lambda_2)$
Remove R_{14}	
mult $R_2R_7 \rightarrow R_7$	$D\ d_{12}p_2^0(\lambda_2)$
mult $R_{10}R_{15} \rightarrow R_{15}$	$F\ d_{12}p_2^0(\lambda_1)$
add $R_7R_{15} \rightarrow R_{15}$	
Remove R_{15}	

Figure 3-4. Storage and Instruction Set

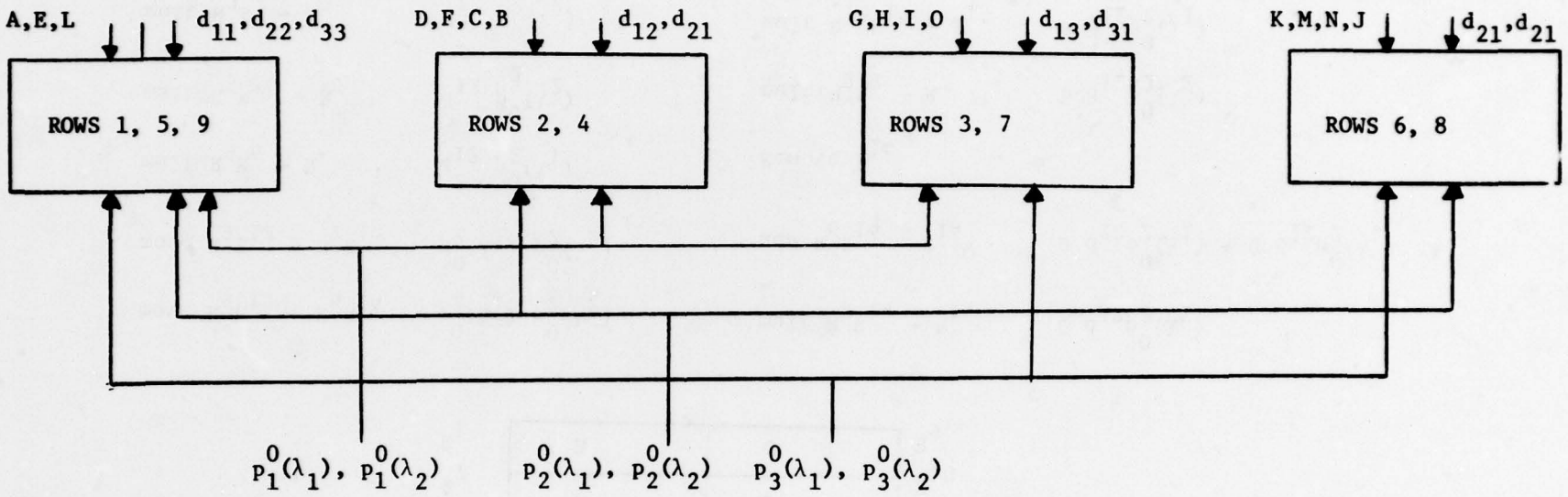


Figure 3-5. Processing array

consider a temporary storage and instruction set shown in Figure 3-4 for rows 2 and 4. With 30 microinstructions, rows 2 and 4 of Figure 3-2 can be formed. Now consider the array shown in Figure 3-5 where all the rows of $q_i^k(\lambda)$ are formed.

The boxes may be considered a bit-sliced ALU, AMD 2901/03, each of which is 4 bits wide. The register set shown previously is a RAM stack 16x4 atop each ALU and part of the ALU monolithic chip. In actuality, the AMD 2901 has a 16x4 RAM stack and no multiply capability. On the other hand, the AMD 2903 has a multiply capability but only an 8x4 RAM stack. Hence our hypothetical bit sliced ALU is a composite of the two and is designated as AMD 2901/03. The instruction set controls all four ALU's in parallel. Since the data is in immediate memory, cycle times are of the order of 200 nanoseconds or less. Hence 30 microinstructions can be executed in 6 microseconds. Done in the parallel fashion as described above, $q_i^k(\lambda)$ can be computed in 6 microseconds using four bit sliced ALU's and one controller.

In summary, we have shown the beginnings of applying bit sliced microprocessors to the relaxation algorithm. An array of four AMD 2901/03 ALU's can compute an intermediate quantity, $q_i^k(\lambda)$, of relaxation in approximately 6 microseconds, with a single instruction set for all four ALU's. Computation for an entire iteration of the two objects by three classes case is probably 8 microseconds or so, and ten iterations could be accomplished in approximately 100 microseconds. We have done this by taking advantage of certain symmetries in the calculations which hold in a number of real cases.

3.1.2 10x100 Case

The mathematical formulation of the relaxation problem is repeated below:

$$p_i^{k+1}(\lambda) = p_i^k(\lambda)[1 + q_i^k(\lambda)]/\sum_\lambda p_i^k(\lambda)[1 + q_i^k(\lambda)]$$

$$q_i^k(\lambda) = \sum_j d_{ij}[\sum_{\lambda'} r_{ij}(\lambda\lambda') p_j^k(\lambda')].$$

We now trace through the formulation, term by term, to estimate the number of operations required by the 10x100 case.

(1) $q_i^k(\lambda)$

For 2 classes and 3 objects, i = 1, 2, 3 and λ = 1, 2 then $q_i^k(\lambda)$ becomes $q_1(\lambda_1)$, $q_2(\lambda_1)$, $q_3(\lambda_1)$, $q_1(\lambda_2)$, $q_2(\lambda_2)$, $q_3(\lambda_3)$. Each is a 3x2 matrix with a sequence of 6 multiplies, then 3 adds, and 3 multiplies; the total is 6x [9 multiplies + 3 adds] = 54 multiplies and 18 adds.

For 10 classes and 100 objects, i = 1, 2, 3, ...100 and λ = 1, 2, ...10, then $q_i^k(\lambda)$ becomes $q_1(\lambda_1)$, $q_2(\lambda_1)$, ...$q_{100}(\lambda_1)$, $q_1(\lambda_2)$, $q_2(\lambda_2)$, ...$q_{100}(\lambda_{10})$, ... Each is a 100 x 10 matrix with a sequence of 1000 multiplies, 100 adds, and 100 multiplies; the total is 100 x 10[1100 multiplies + 100 adds] = 1,100,000 multiplies and 100,000 adds.

(2) $\sum_\lambda p_i^k(\lambda)[1 + q_i^k(\lambda)]$

For 2 classes and 3 objects i = 1, 2, 3, and λ = 1, 2, then

$$\sum_\lambda p_i^k(\lambda)[1 + q_i^k(\lambda)] = p_i^k(\lambda_1)[1 + q_i^k(\lambda_1)] + p_i^k(\lambda_2)[1 + q_1(\lambda_2)].$$

That is, for each i, we sum over λ. Each i is composed of a sequence of 2 adds, 2 multiplies, and 1 add, for a total of 3 adds and 2 multiplies. This is repeated 3 times for 9 adds and 6 multiplies.

For 10 classes and 100 objects, each i is composed of a sequence of 10 adds, 10 multiplies, and 10 adds for a total of 20 adds and 10 multiplies. This is repeated 100 times for 2000 adds and 1000 multiplies.

(3) $q_i^k(\lambda)[1 + q_i^k(\lambda)]$

For 2 classes and 3 objects, there are 6 combinations of 1 add and 1 multiply for a total of 6 adds and 6 multiplies.

For 10 classes and 100 objects, there are 1000 adds and 1000 multiplies.

(4) $p_i^k(\lambda) = p_i^k(\lambda)[1 + q_i^k(\lambda)]/\sum_\lambda p_i^i(\lambda)[1 + q_i^k(\lambda)]$

For 3 objects and 2 classes, there are 6 combinations for $p_i^k(\lambda)$, therefore, 6 divides.

For 10 classes and 100 objects, there are 1000 divides. In summary, for 10 classes and 100 objects, there are

(1) 1.1×10^6 multiplies

1×10^5 adds

(2) 2×10^3 adds

1×10^3 multiplies

(3) 1×10^3 adds

1×10^3 multiplies

(4) 1×10^3 divides.

3.2 Connected Components

The Connected Components Algorithm is a so-called segmentation algorithm, i.e. it is attempting to separate out those portions of the image which are regarded as possible targets. Since the image is thresholded and only those portions of the image which exceed threshold remain and are labelled as 1's, the algorithm is regarded as binary in nature. Of course, the algorithm is repeated at a number of thresholds to find the best match with the edges. Nevertheless at each threshold the algorithm may be regarded as binary. Let us pause for a moment in the hardware implementation and discuss several broad approaches to implementation and the hardware impacts of each.

With a binary algorithm, one naturally thinks immediately of a bit-plane approach. That is, each pixel is represented by a processing module in an array of modules and each module looks at its immediately surrounding neighbors to determine connectedness. While this approach has captured the fancy of a number of theoretical investigators, it has not kindled the same degree of enthusiasm among hardware developers because of state of the art limitations. On the other hand, while hardware developers talk about pipeline operations, shift registers for memory and hardware-wired processors for each algorithms, theoretical investigators view these as woefully inadequate in terms of speed. Interestingly enough, the Connected Components Algorithm provides an appropriate vehicle to attempt to land somewhere between these two doctrines.

More specifically, the binary data stream will enter the algorithm and emerge transformed into different signal levels for different regions. Readout of the binary picture will progress one horizontal line at a time starting with the top line and progressing downward. Each horizontal line will be read out starting from left to right. Since the image data is read out serially, the algorithm is a local operator. The labelling of each non-zero element in the image plane could be done as shown in Figure 3-6.

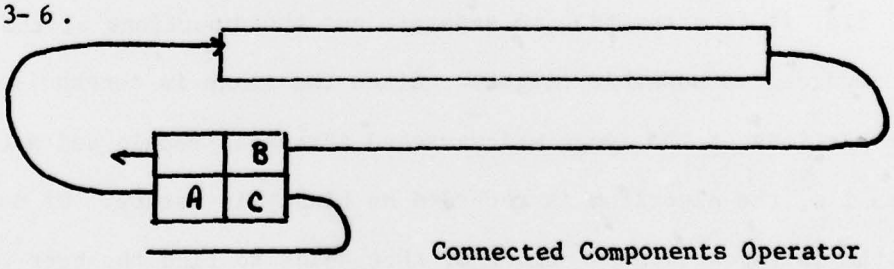


Figure 3-6. Data Flow through Connected Components Operator

The Operator is a transformation from a binary picture to a colored one by a mapping T:

$$T(A, B, M): \quad C \qquad C^1$$

where C^1 is the color of the transformed pixel C, the variables A and B represent nearest neighbors of C, and M represents an available color. The relative locations of pixels A, B, and C in the image plane are shown in Figure 3-7.

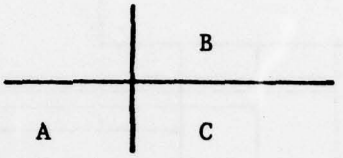


Figure 3-7. Relative Location of A,B,C

We define the coloring window as always containing these elements. Elements A and B are nearest neighbors of C and have already been processed by the Operator. Element B is located one horizontal line above elements C and A. Element C is painted by the Operator according to the following rule:

For $C \neq 0$

$$C^1 \quad \begin{cases} A \text{ if } A \neq 0,\ B \neq 0 \\ B \text{ if } A = 0,\ B \neq 0 \\ M \text{ if } A = 0,\ B = 0 \end{cases}$$

When adjacent elements have different colors, the element being painted assumes the color of the nearest neighbor in the same line (rows dominate). Whenever elements A and B are zero and element C is not zero, element C^1 is given a new color. Multi-colored object regions occur when pixels A and B are both non-zero, not equal to each other, and C is non-zero, i.e.,

For $C \neq 0$, $A \neq B \neq 0$.

The basic ALU we have chosen to use is 4 bits wide; it seems too wide for the present application unless we could convert it to 4 processors, each one bit wide.

The AMD 2901 outputs data from the ALU as shown in Figure 3-8 , i.e. the individual bits are

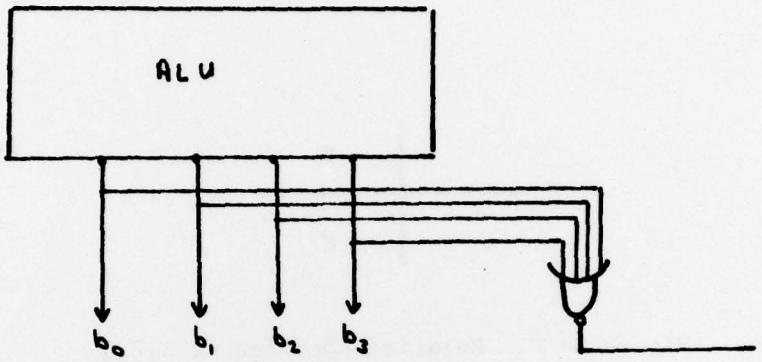


Figure 3-8. Zero Detect Logic

available as well as a zero detect flag. One approach is to zero detect on the individual bits, and use them to control a particular area of memory in which the segments, each corresponding to a particular threshold, are stored. Then as we proceed with the branching logic of the Connected Components Operator, those bit positions which exited early are masked for subsequent branches. This approach calls for more hardware, which is a disadvantage.

Another approach is to use microinstructions to shift the bits out of a holding register, as in the AMD 2903, one at a time, mask them and zero detect. Here, of course, the processing time is longer but the hardware is not increased. The key to this is exactly how the ALU performs the AND functions and the NOT EQUAL ZERO function. In the Advanced Micro Devices ALU's, the bits are individually AND'ed and the four bits presented to an output register. If they are also zero, a ZERO DETECT flag is set. Let us assume for the moment that quantities A, B, and C

are located in the RAM (see the Arithmetic Logic Unit discussion in the first quarterly report) above the ALU. According to the logic just described for the Connected Components Operator, the first test is $C \neq 0$. Since we are operating in four independent modes, there doesn't seem to be any advantage to obtaining a ZERO DETECT bit across all four. The second test is $A \neq 0$; if $A = 0$, then a test on $B \neq 0$ must be performed. The problem is that a decision tree must be executed which operates for four different bits or threshold levels; therefore, each may exit at a different branch point. However, the entire process of operating at four different levels simultaneously is controlled by a single set of instructions. So, more clearly stated, the problem is to operate at four different levels with a single instruction string, allocating one bit per level. Yet another approach is to concatenate the bits from a number of the thresholds and use them as an address for a table look up via a direct memory access. For four thresholds, a 4096 x 12 bit EPROM would be required. The drawback here is board space. A memory of 164 x 16 plus support logic would require a 9 x 12 board.

A final approach, which is another hardware addition, is to use an LSI logic array such as a 12-input Field Programmable Logic Array (FPLA) and generate logic signals from an array of 3 ALU's.

In the next quarter, we will analyze these approaches in more detail with the goal of determining the appropriate architecture for our image processing module.

4.0 IMAGE PROCESSOR ARCHITECTURE

The purpose of this section is to generalize and describe the architecture as discussed in the prior sections. The architecture developed follows along the lines of the University of Maryland algorithms for Image Processing Using Overlays. We shall address ourselves to specific component parts of the processor such as registers, I/O arrays, memory, and image processing module (ALU).

4.1 Registers

It appears that algorithms like relaxation will require a substantial register set above the ALU and part of the monolithic chip. It also appears likely that some flexibility is desired in word length within the register. More specifically, individual bit position addressing seems to offer some advantages. The problem here is that the number of control lines will increase as word length descreases. Further, some algorithms will require a particular word length, and other algorithms will be more appropriately implemented with another word length.

4.2 Input/Output

The course to follow here seems to be along the lines of the instruction fetch microinstruction as already implemented in commercial microprocessors. Namely, they fetch the next instruction while the current instruction is being executed. With regard to the registers, it would be desirable to be able to load them while the ALU is executing a prior instruction. The Texas Instruments TI 74SN 481 described in Section 2.3 of the first quarterly goes further and has multiple use ALU which increments registers while the ALU is performing some other operation. The TI chip has a large number of control lines which facilitates these multiple operations.

4.3 Arrays

There has been a substantial amount of interest in microprocessor

arrays which seem to vary from each bit slice unit representing a single pixel to each bit slice representing an arbitrary pixel within the image to a fixed array of bit slice units cascaded with enough word length to perform high speed, parallel operations. Although this is a relatively new technical field with few constraints, there are a few statements which can be made. The notion of having each bit slice unit represent a single pixel within a image of say 500 x 600 pixels for an array of 300,000 bit slice units seems unwieldy and cumbersome, to put it mildly. On the other hand, a fixed array of microprocessors which breaks up an algorithm and executes it in a single instruction-multiple data fashion (as we described in the first quarterly report in regard to the implementation of the 2 x 3 non-linear probabilistic relaxation case) does not offer sufficient flexibility to handle the Connected Components Operator, for example. If we take the approach that the array of microprocessors are merely resources that are to be organized in a manner commensurate with several different image processing algorithm structures and image features, the problem becomes much more difficult but the solution more general. Then in this area, we shall be looking for techniques to render the interconnects as flexible as possible.

4.4 Memory

The idea here is that the presence of an ALU does not preclude the use of memories such as EPROMs for look up tables. This is particularly true if one look up table can replace 12-15 microinstructions per cycle. Secondly, memory may be useful where data is available a number of cycles before the microinstruction execution which will use it. More immediate data will be kept in the registers atop the ALU. Finally, in providing hardware implementation for the University of Maryland, Westinghouse has consistently promoted the idea of developing algorithms which do not require full frame storage. It may turn out that in the more general case, full frame storage is required and we should be aware of it.

4.5 Image Processing Module (ALU)

We have decided to label the arithmetic logic units (ALU) as an image processing module because it appears, based on six months work on this project contract, that the architecture will be substantially different from that of the ALU's described in Section 2.3 of the first quarterly report. Certainly, we shall be including multiply and divide requirements on the monolithic chip. Further, it appears likely that the image processing module must be capable of handling a flexible word length. Finally, the module may in fact require some sort of a gated logic array following the output register to facilitate the flexible word length.

REFERENCES

1. A Computer Control Unit Using the AM 2909, J.R.W. Clymer, Applications Note, Advanced Micro Devices, Inc., Sunnyvale, CA., 1976.

2. Microprogramming Hnadbook, J.R. Mick and J. Brick, Applications Note, Advanced Micro Devices, Inc., Sunnyvale, CA., 1976

3. How Bit-Slice Families Compare: Part 2, Sizing Up the Microcontrollers, W.T. Adams and S.M. Smith, Electronics, Aug. 17, 1978, McGraw-Hill Publications, New York, New York.

4. Postscript On Bit-Slice Families: Microcontrollers Serve Many Needs, W.T. Adams and S.M. Smith, Electronics, Aug. 31, 1978, McGraw-Hill Publications, New York, New York.